

CHAPTER

7

Advanced Shell Programming

case ► Dominion Consulting's managers are pleased with the new shell program that maintains the employee phone file. Based on the relatively small volume of transactions and file size, the programming supervisor agrees that the shell program will be a permanent solution for maintaining the corporate phone file. However, you need to enhance the program and incorporate new features, such as preventing the same phone number from being assigned to more than one person, re-entering data to correct errors, and sorting by employees' last names.

LESSON A

objectives

In this lesson you will:

- Use flowcharting and pseudocode tools
- Learn to write scripts that tell the system which shell to use as an interpreter
- Use the test command to compare values and validate file existence
- Use the translate command, tr, to display a record with duplicate fields
- Use the sed command to delete a phone record

Developing a Fully Featured Program

In the last chapter you began developing a program to automate the maintenance of the phone records in the corp_phones file. You developed a menu that presents several options and a data entry screen that allows records to be added to the file.

In this chapter you complete the program. You add code that deletes a specified record from the file, searches for a specified record, and sorts and displays all records in the file. You perform these tasks with the test, grep, sed, and tr commands. Before you begin adding this code, you learn about two standard program development tools: flowcharts and pseudocode.

Analyzing the Program

A computer program is developed by analyzing the best way to achieve the desired results. Standard programming analysis tools help you do this. The two most popular and proven analysis tools are the program flowchart and pseudocode.

Flowcharting

The **flowchart** is a logic diagram that uses a set of standard symbols that explain the program's sequence and each action it takes. For example, look at the flowchart in Figure 7-1.

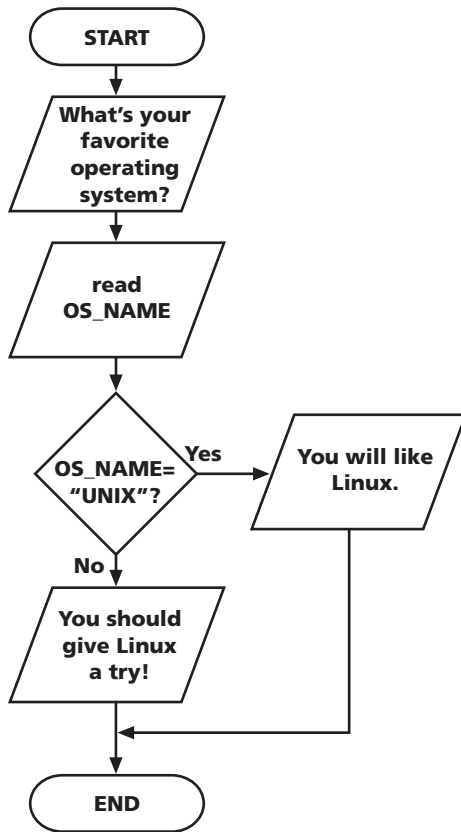


Figure 7-1: Sample flowchart

Figure 7-1 is a flowchart for the following program, which you wrote in Chapter 6.

```
echo -n "What is your favorite operating system? "  
read OS_NAME  
if [ "$OS_NAME" = "UNIX" ]  
then  
    echo "You will like Linux."  
else  
    echo "You should give Linux a try!"  
fi
```

Each step in the program is represented by a symbol in the flowchart. The shape of the symbol indicates the type of operation being performed, such as input/output or a decision. Figure 7-2 shows standard flowcharting symbols and their meanings.

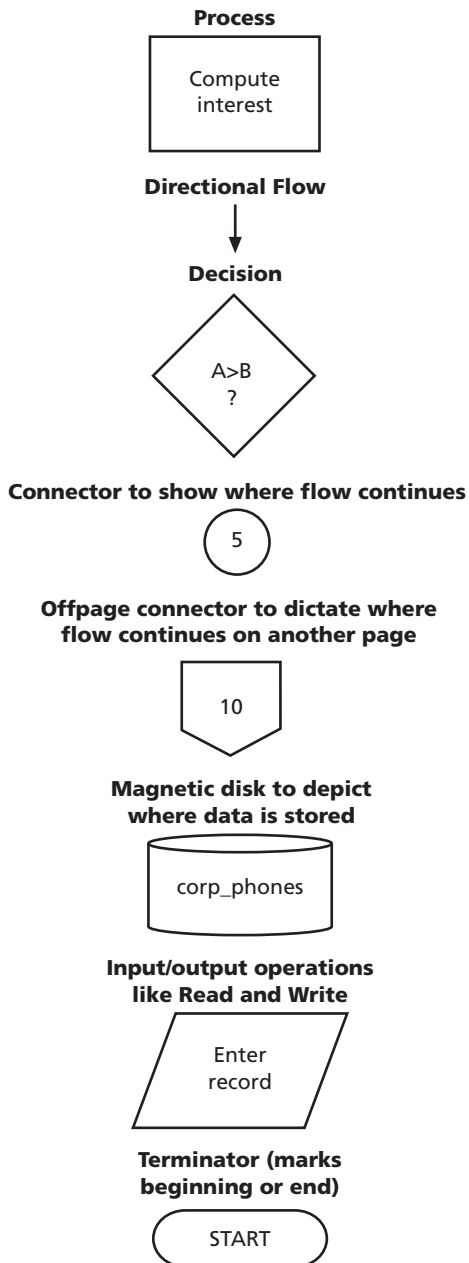


Figure 7-2: Standard flowchart symbols

The arrows that connect the symbols represent the direction in which the program flows from one symbol to the next. In the flowchart in Figure 7-1, the arrow after the START terminator shows the program flowing to an output operation that displays the message, “What is your favorite operating system?” Next, the program flows to an input operation that reads a value into OS_NAME. A decision structure (represented by a diamond-shaped symbol) is encountered next, which compares OS_NAME to “UNIX” to determine if the two are equal. If so, the program follows the “Yes” branch. This leads to an output operation that displays the message, “You will like Linux.” If OS_NAME and “UNIX” are not equal, the program follows the “No” branch. This leads to an output operation that displays the message, “You should give Linux a try!” The two paths then converge and flow to the END terminator.

You manually create a flowchart using a drawing template. Flowchart templates provide the symbols that denote logical structures, input-output operations, processing operations, and the storage media that contain the files. (See Figure 7-2.) A variety of flowcharting software packages let you create flowcharts on your computer. Popular word-processing packages, such as Microsoft Word and WordPerfect, are also equipped with flowcharting tools.

Writing Pseudocode

After creating a flowchart, the next step in designing a program is to write **pseudocode**. Pseudocode instructions are similar to actual programming statements. Use them to create a model that you can later use as a basis for a real program. For example, here are pseudocode statements for the `os_choice` program:

```
Display "What is your favorite operating system? " on the
screen.
Enter data into OS_NAME.
If OS_NAME is equal to "UNIX"
    then
        Display "You will like Linux." on the screen.
Else
    Display "You should give Linux a try!" on the screen.
End if.
```

Pseudocode is a design tool only, and never processed by the computer. Therefore, there are no strict rules to follow. The pseudocode should verbally match the symbolic representation of logic illustrated on the flowchart. For example, Figure 7-3 shows the flowchart and pseudocode that represent one change Dominion Consulting wants to make to its phone program (re-entering data in a field).

Before you can begin to work on the program, however, you need to do some preliminary work. First, you must ensure that everyone who runs the program does so with the correct shell. To do this, modify the program so it informs the operating system which shell to use while interpreting the statements in the script.

Pseudocode:

```

While entry = minus
do
    Reposition cursor in previous field.
    Enter the data.
    Reposition cursor on field following the field that was
        just entered (re-entered data)
    Continue testing while loop.
Done

```

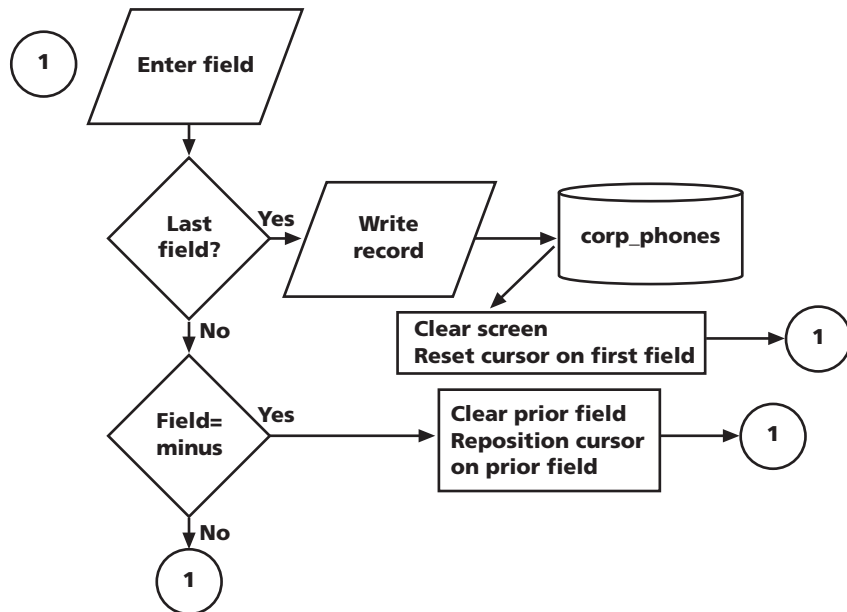
Flowchart:

Figure 7-3: Pseudocode and flowchart for re-entry of previous fields

Ensuring the Correct Shell Runs the Script

Each UNIX user has the freedom and capability of choosing which shell he or she prefers. When developing a shell script, you must be responsible for ensuring that the correct shell is used to interpret your script. This is because all shells do not support the same commands and programming statements.

You can instruct the system to run a script with a specific shell. You do so in the first line of the script. The line must start with the # character, followed by the

! character and the path of the shell. For example, this line tells the system to use the Bash shell:

```
#!/bin/bash
```

When the system reads this code line, it loads the Bash shell and uses it to interpret the statements in the script file. Since UNIX includes many shells, you should always begin your scripts with this statement.

Next, you need to make sure that the necessary directories and files are in place before you execute programs that depend on them. To do this, use the test command.

Using the test Command

The **test** command makes preliminary checks of the UNIX internal environment and other useful comparisons (beyond those that the if command alone can perform). See Appendix B, “Syntax Guide to UNIX Commands,” for a description of options available for the test command. You can place the test command inside your shell script or execute it directly from the command line. In this section you learn to use the test command at the command line. In the next section you use it in a script file.

The test command uses operators expressed as options to perform the evaluations. The command can be used to:

- Perform relational tests with integers (such as equal, greater than, less than, etc.)
- Test strings
- Determine if a file exists and what type of file it is
- Perform Boolean tests

Each type of test operation is discussed in more detail in the following sections.

Relational Integer Tests with the test Command

The test command can determine if one integer is equal to, greater than, less than, greater than or equal to, less than or equal to, or not equal to another integer. Table 7-1 describes the integer testing options of the test command.

Option	Meaning	Example Command
-eq	equal to	test a -eq b
-gt	greater than	test a -gt b
-lt	less than	test a -lt b
-ge	greater than or equal to	test a -ge b
-le	less than or equal to	test a -le b
-ne	not equal to	test a -ne b

Table 7-1: Integer testing options of test command

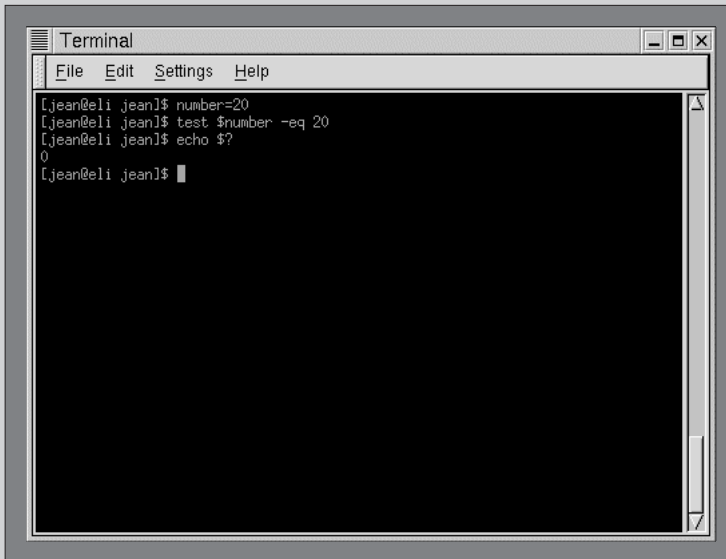
The `test` command returns a value known as an exit status. An **exit status** is a numeric value that the command returns to the operating system when it finishes. The value of the `test` command's exit status indicates the results of the test performed. If the exit status is 0 (zero), the test result is true. An exit status of 1 indicates the test result is false.

The exit status is normally detected in a script by the `if` statement or in a looping structure. You can view the last command's exit status by typing the command:

```
echo $?
```

To demonstrate the `test` command with integer expressions:

- 1 Create the variable `number` with the value 20 by typing the command `number=20` and pressing **Enter**.
- 2 Type `test $number -eq 20` and press **Enter**.
- 3 Type `echo $?` and press **Enter**. Your screen looks similar to Figure 7-4.

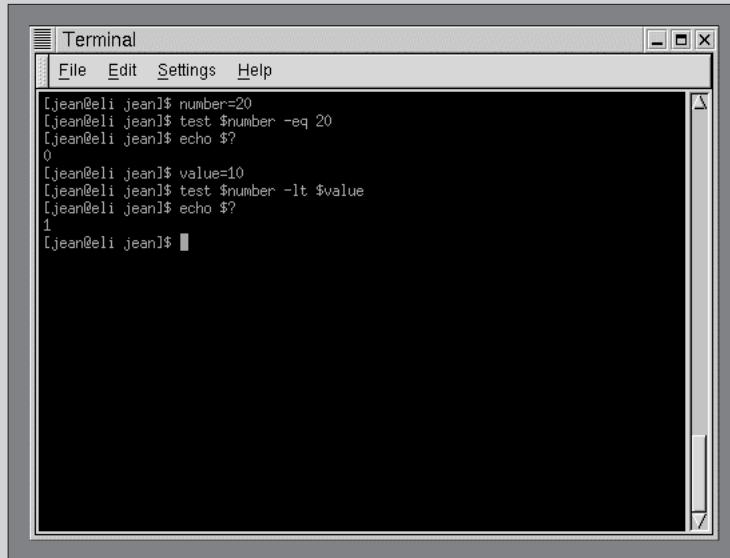


```
Terminal
File Edit Settings Help
[jean@eli jean]$ number=20
[jean@eli jean]$ test $number -eq 20
[jean@eli jean]$ echo $?
0
[jean@eli jean]$
```

Figure 7-4: Testing an integer found true

The echo `$?` command displays the exit status of the last command that was executed. In this example, the test command returns the exit status 0, indicating the expression `$number -eq 20` is true. This means `$number` is equal to 20.

- 4 Type **value=10** and press **Enter**.
- 5 Type **test \$number -lt \$value** and press **Enter**.
- 6 Type **echo \$?** and press **Enter**. Your screen looks similar to Figure 7-5.



```
Terminal
File Edit Settings Help
[jean@eli jean]$ number=20
[jean@eli jean]$ test $number -eq 20
[jean@eli jean]$ echo $?
0
[jean@eli jean]$ value=10
[jean@eli jean]$ test $number -lt $value
[jean@eli jean]$ echo $?
1
[jean@eli jean]$
```

Figure 7-5: Results of test command

In this example, the test command returns the exit status 1, indicating the expression `$number -lt $value` is false. This means `$number` is not less than `$value`.

String Tests with the test Command

You can use the test command to determine if a string has a length of zero characters or a non-zero number of characters. It can also test two strings to determine if they are equal or not equal. Table 7-2 describes the string testing options of the test command.

Option or Expression	Meaning	Example
-z	Tests for a zero-length string	test -z string
-n	Tests for a non-zero string length	test -n string
string1 = string2	Tests two strings for equality	test string1 = string2
string1 != string2	Tests two strings for inequality	test string1 != string2
string	Tests for a non-zero string length	test string

Table 7-2: String testing options of test command

To demonstrate the test command's string testing capabilities:

- 1 Type **name="Bjorn"** and press **Enter**.
- 2 Type **test \$name = "Bjorn"** and press **Enter**.
- 3 Type **echo \$?** and press **Enter**. Your screen looks similar to Figure 7-6.

```

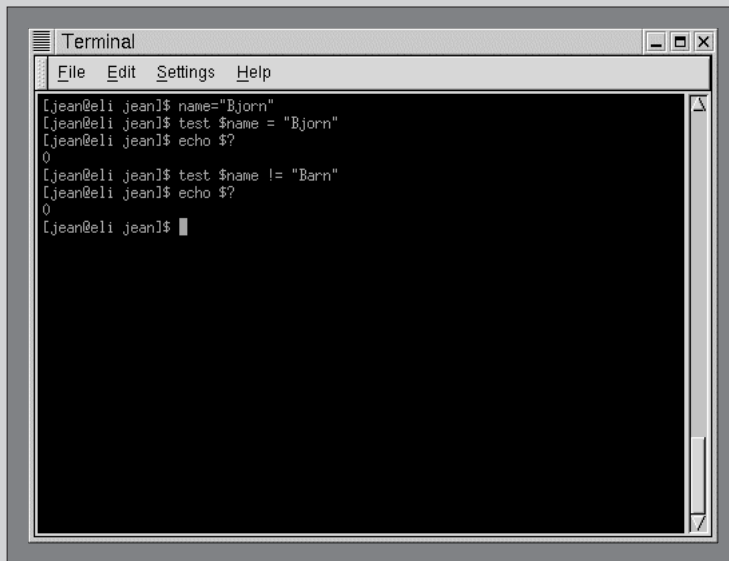
Terminal
File Edit Settings Help
[jean@eli jean]$ name="Bjorn"
[jean@eli jean]$ test $name = "Bjorn"
[jean@eli jean]$ echo $?
0
[jean@eli jean]$

```

Figure 7-6: Testing a string found true

In the example above, the test command returns the exit status 0 indicating the expression `$name = "Bjorn"` is true. This means `$name` and "Bjorn" are equal.

- 4 Type **test \$name != "Barn"** and press **Enter**.
- 5 Type **echo \$?** and press **Enter**. Your screen looks similar to Figure 7-7.

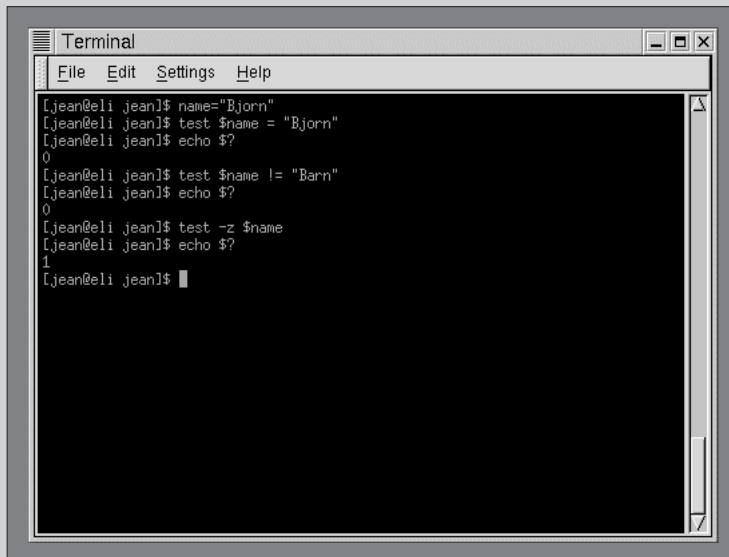
A terminal window titled "Terminal" with a menu bar (File, Edit, Settings, Help). The terminal shows the following commands and output:

```
[jean@eli jean]$ name="Bjorn"
[jean@eli jean]$ test $name = "Bjorn"
[jean@eli jean]$ echo $?
0
[jean@eli jean]$ test $name != "Barn"
[jean@eli jean]$ echo $?
0
[jean@eli jean]$
```

Figure 7-7: Testing unequal strings

In the example above, the test command returns the exit status 0 indicating the expression `$name != "Barn"` is true. This means `$name` and "Barn" are not equal.

- 6 Type **test -z \$name** and press **Enter**.
- 7 Type **echo \$?** and press **Enter**. Your screen looks similar to Figure 7-8.

A terminal window titled "Terminal" with a menu bar (File, Edit, Settings, Help). The terminal shows the following commands and output:

```
[jean@eli jean]$ name="Bjorn"
[jean@eli jean]$ test $name = "Bjorn"
[jean@eli jean]$ echo $?
0
[jean@eli jean]$ test $name != "Barn"
[jean@eli jean]$ echo $?
0
[jean@eli jean]$ test -z $name
[jean@eli jean]$ echo $?
1
[jean@eli jean]$
```

Figure 7-8: Testing a string found false

The last test command in the example above returns the exit status 1 indicating the expression `-z $name` is false. This means the string `$name` is not zero length.

Testing Files with the test Command

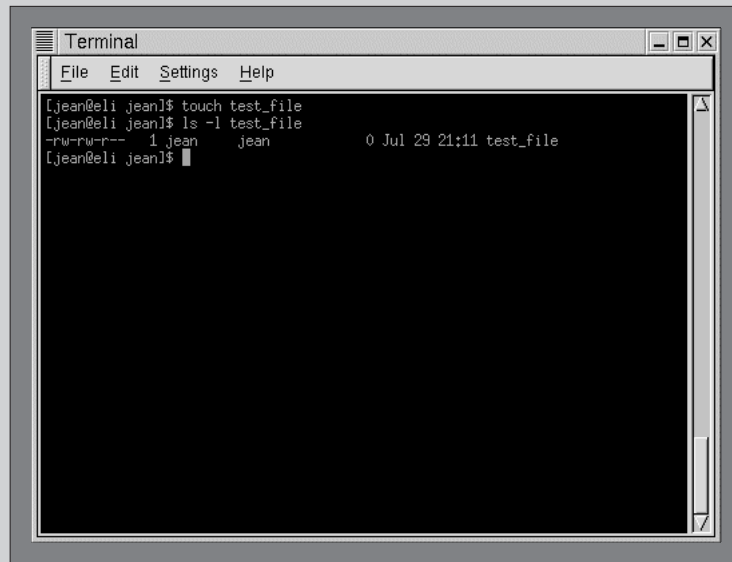
The test command can determine if a file exists and if it has a specified permission or attribute (such as executable, readable, writeable, directory, etc.). Table 7-3 describes several of the command's file testing options.

Option	Meaning	Example
-e	True if a file exists	test -e file
-r	True if a file exists and is readable	test -r file
-w	True if a file exists and is writeable	test -w file
-x	True if a file exists and is executable	test -x file
-d	True if a file exists and is a directory	test -d file
-f	Tests if a file exists and is a regular file	test -f file
-s	True if a file exists and has a size greater than zero	test -s file
-c	Tests if a file exists and is a character special file (which is a character-oriented device, such as a terminal or printer)	test -c file
-b	Tests if a file exists and is a block special file (which is a block-oriented device, such as a disk or tape drive)	test -b file

Table 7-3: File testing options of test command

To demonstrate the test command's file testing capabilities:

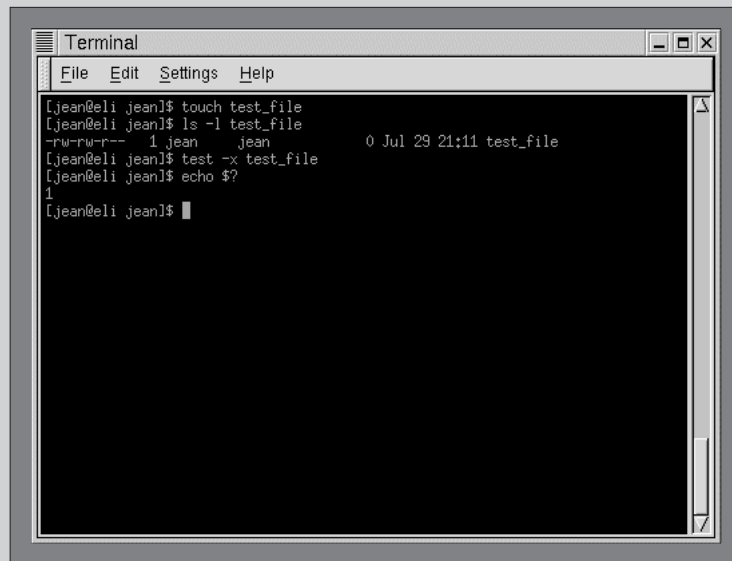
- 1 Use the `touch` command to create an empty file named `test_file`.
- 2 Use the `ls -l test_file` command to view the file's permissions. Your screen should look similar to Figure 7-9.
- 3 Note that the file has read and write permissions for you, the owner. Then type `test -x test_file` and press **Enter**.

A terminal window titled "Terminal" with a menu bar (File, Edit, Settings, Help). The command history shows: [jean@eli jean]\$ touch test_file, [jean@eli jean]\$ ls -l test_file, and the output: -rw-rw-r-- 1 jean jean 0 Jul 29 21:11 test_file. The prompt [jean@eli jean]\$ is at the bottom.

```
Terminal
File Edit Settings Help
[jean@eli jean]$ touch test_file
[jean@eli jean]$ ls -l test_file
-rw-rw-r-- 1 jean jean 0 Jul 29 21:11 test_file
[jean@eli jean]$
```

Figure 7-9: Creating test_file file

- 4 Type **echo \$?** and press **Enter**. Your screen looks similar to Figure 7-10.

A terminal window titled "Terminal" with a menu bar (File, Edit, Settings, Help). The command history shows: [jean@eli jean]\$ touch test_file, [jean@eli jean]\$ ls -l test_file, [jean@eli jean]\$ test -x test_file, [jean@eli jean]\$ echo \$?, and the output: 1. The prompt [jean@eli jean]\$ is at the bottom.

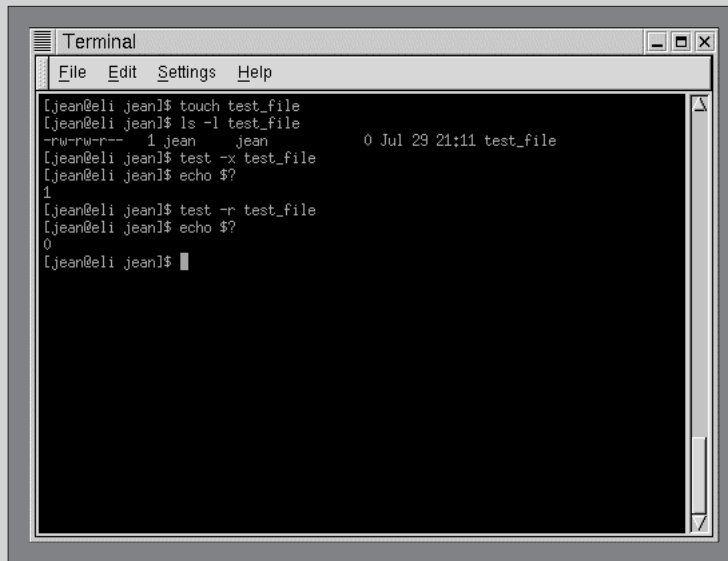
```
Terminal
File Edit Settings Help
[jean@eli jean]$ touch test_file
[jean@eli jean]$ ls -l test_file
-rw-rw-r-- 1 jean jean 0 Jul 29 21:11 test_file
[jean@eli jean]$ test -x test_file
[jean@eli jean]$ echo $?
1
[jean@eli jean]$
```

Figure 7-10: Testing a file to see if it's executable

The test command returns an exit status of 1, because test_file is not executable.

- 5 Type **test -r test_file** and press **Enter**.

- 6** Type `echo $?` and press **Enter**. Your screen looks similar to Figure 7-11.



```

Terminal
File Edit Settings Help
[jean@eli jean]$ touch test_file
[jean@eli jean]$ ls -l test_file
-rw-rw-r-- 1 jean jean 0 Jul 29 21:11 test_file
[jean@eli jean]$ test -x test_file
[jean@eli jean]$ echo $?
1
[jean@eli jean]$ test -r test_file
[jean@eli jean]$ echo $?
0
[jean@eli jean]$

```

Figure 7-11: Testing a file to see if it's readable

The test command returns an exit status of 0 indicating test_file is readable.

Performing Boolean Tests with the test Command

The test command's Boolean operators let you combine multiple expressions with AND and OR relationships. You can also use a Boolean negation operator. Table 7-4 describes Boolean operators.

Option	Meaning	Example
-a	Logical AND	test expression1 -a expression2
-o	Logical OR	test expression1 -o expression2
!	Logical negation	test !expression

Table 7-4: Test command's Boolean operators

The `-a` operator combines two expressions and tests a logical AND relationship between them. The form of the test command with the `-a` option is:

```
test expression1 -a expression2
```

If both `expression1` and `expression2` are true, the `test` command returns true (with a 0 exit status). However, if either `expression1` or `expression2` is false, the `test` command returns false (with an exit status of 1).

The `-o` operator also combines two expressions. It tests a logical OR relationship. The form of the `test` command with the `-o` option is:

```
test expression1 -o expression2
```

If either `expression1` or `expression2` is true, the `test` command returns true (with a 0 exit status). However, if neither of the expressions is true, the `test` command returns false (with an exit status of 1).

The `!` operator negates the value of an expression. This means that if the expression normally causes `test` to return true, it returns false instead. Likewise, if the expression normally causes `test` to return false, it returns true instead. The form of the `test` command with the `!` operator is:

```
test !expression
```

To demonstrate the `test` command's Boolean operators:

- 1 Recall that the `test_file` file you created in the previous exercise has read and write permissions. Type **`test -r test_file -a -w test_file`** and press **Enter**.
This command tests two expressions using an AND relationship: `-r test_file` and `-w test_file`. If both expressions are true, the `test` command returns true.
- 2 Type **`echo $?`** and press **Enter**. Your screen looks similar to Figure 7-12.

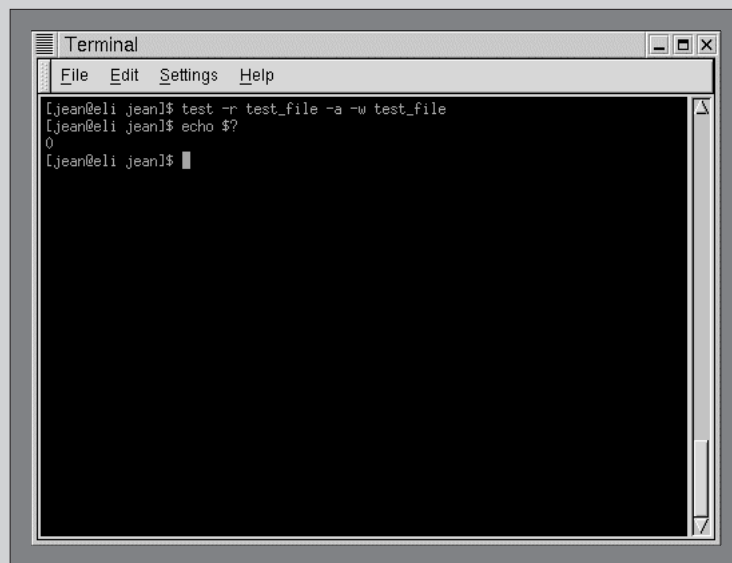


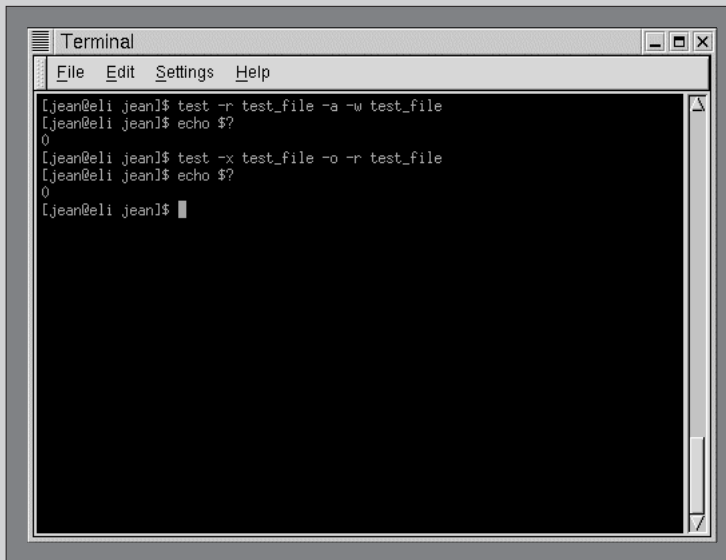
Figure 7-12: Using Boolean operators to test a file

The test command returns an exit status of 0 indicating that test_file is readable and writable.

- 3** Type **test -x test_file -o -r test_file** and press **Enter**.

This command tests two expressions using an OR relationship: -x test_file and -r test_file. If either of these expressions is true, the test command returns true.

- 4** Type **echo \$?** and press **Enter**. Your screen looks similar to Figure 7-13.

A terminal window titled "Terminal" with a menu bar (File, Edit, Settings, Help). The terminal shows the following commands and output:

```
[jean@eli jean]$ test -r test_file -a -w test_file
[jean@eli jean]$ echo $?
0
[jean@eli jean]$ test -x test_file -o -r test_file
[jean@eli jean]$ echo $?
0
[jean@eli jean]$
```

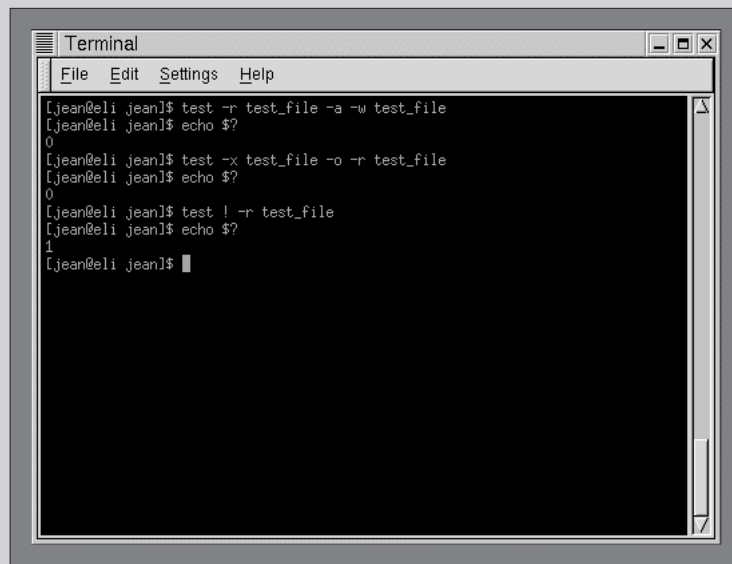
Figure 7-13: Testing with OR operator

The test command returns an exit status of 0 indicating that test_file is either executable OR readable.

- 5** Type **test ! -r test_file** and press **Enter**.

This command negates the result of the expression -r test_file. If the expression is true, the test command returns false. Likewise, if the expression is false, the test command returns true.

- 6** Type **echo \$?** and press **Enter**. Your screen looks similar to Figure 7-14.

A terminal window titled "Terminal" with a menu bar (File, Edit, Settings, Help). The terminal shows a series of commands and their outputs. The first command is `test -r test_file -a -w test_file`, which returns an exit status of 0. The second command is `test -x test_file -o -r test_file`, which also returns an exit status of 0. The third command is `test ! -r test_file`, which returns an exit status of 1. The user's prompt is `[jean@eli jean]$` and the output is `0` or `1` followed by a prompt character.

```
[jean@eli jean]$ test -r test_file -a -w test_file
[jean@eli jean]$ echo $?
0
[jean@eli jean]$ test -x test_file -o -r test_file
[jean@eli jean]$ echo $?
0
[jean@eli jean]$ test ! -r test_file
[jean@eli jean]$ echo $?
1
[jean@eli jean]$
```

Figure 7-14: Testing with negation operator

The test command returns an exit status of 1 indicating the expression `! -r test_file` is false.

You next use the test command to determine if a directory exists. This lets you set up your environment properly to run the shell scripts you complete in Lesson B. In the following exercise you determine if you have a source directory in your home directory. If not, you create one and add it to the PATH variable. This enables you to run script files stored there without having to type `./` before their names.

To determine if a directory exists:

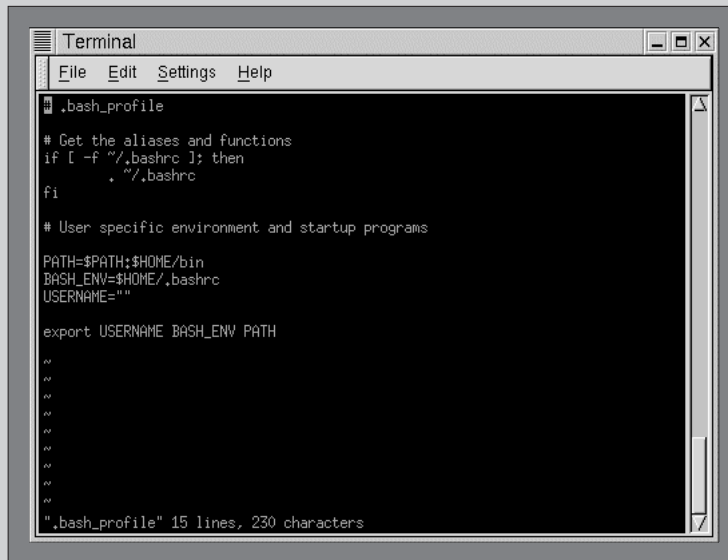
- 1** Type `test -d source ; echo $?` and press **Enter**. This command determines if source exists and if it is a directory. Because the `echo $?` command is included on the same line, the exit status appears immediately after you press **Enter**. If the exit status is 0, you already have a source directory. If this is so, skip to Step 3. Otherwise, continue to Step 2.
- 2** If the command you entered in Step 1 results in exit status 1, you must create the /source directory. Type `mkdir source` and press **Enter**.
- 3** In Chapter 6 you created the file `corp_phones` and the shell scripts `phmenu` and `phoneadd`. Determine if you have the `corp_phones` file by typing `test -e corp_phones ; echo $?` and pressing **Enter**. If you see the exit status 0, the file exists.

help

- 4 Repeat the test command for the phmenu, phoneadd and phlist1 files. After you confirm that these files exist, copy them to the source directory.

If you do not have the files from Chapter 6, see your instructor or technical support person for assistance.

- 5 If you permanently add the /home/user-name/source (where user-name is your login name) directory to your PATH variable, it takes effect each time you log on. You should do this in your .bash_profile file. Load the file into the vi editor. Your screen looks similar to Figure 7-15.



```
Terminal
File Edit Settings Help

~ .bash_profile

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
    . ~/.bashrc
fi

# User specific environment and startup programs

PATH=$PATH:$HOME/bin
BASH_ENV=$HOME/.bashrc
USERNAME=""

export USERNAME BASH_ENV PATH

"~/.bash_profile" 15 lines, 230 characters
```

Figure 7-15: .bash_profile file

- 6 Move the cursor to the line that reads:
`PATH=$PATH:$HOME/bin`
- 7 Type **i** to switch to insert mode. Type `:$HOME/source` (include the colon) at the end of the line.
- 8 Press **Esc**, then type `:wq` and press **Enter** to save the file and exit the editor.
- 9 To make the new PATH value take effect, you must log off and then log back on. Do so now.

You created the source directory and added it to your PATH variable. You can store your script files there and execute them just by typing their names at the command line. In the next exercise, you validate the existence of the corp_phones file.

To validate the existence of a file:

- 1** Make sure you are in your source directory by typing `$ cd ~/source` and then pressing the **Enter** key.
- 2** Type `test -e corp_phones` and then press **Enter**.
- 3** Type `echo $?` and then press **Enter**. Your screen should show a “0,” which indicates a true status.

To see a demonstration of a false status, repeat the test for the file `phone_corp`, which does not exist.

Next you practice using the test command in a script file. Once again, recall the `os_choice` script you wrote in Chapter 6:

```
echo -n "What is your favorite operating system? "
read OS_NAME
if [ "$OS_NAME" = "UNIX" ]
then
    echo "You will like Linux."
else
    echo "You should give Linux a try!"
fi
```

In the following exercise, you will modify the if statement so it uses the test command. When done, the program runs identically as it did before.

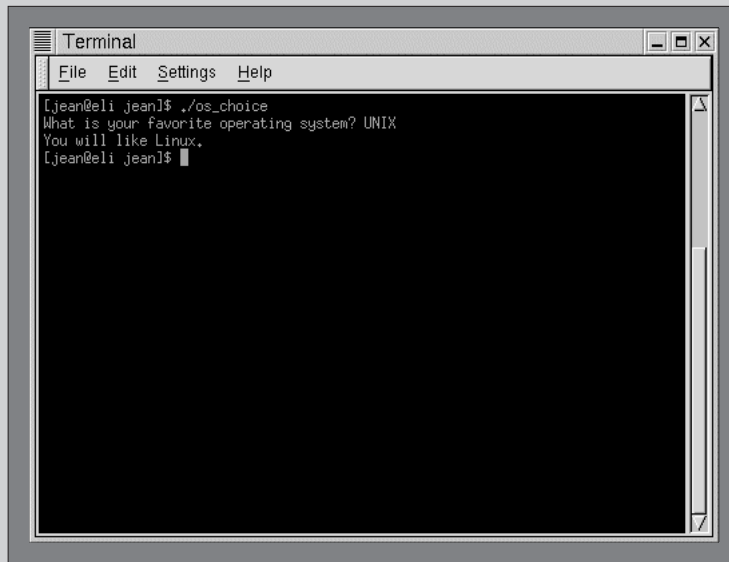
To modify the if statement to use the test command:

- 1** Make sure you are in your home directory. Load the `os_choice` file into vi or Emacs.
- 2** Change the line that reads:

```
if [ "$OS_NAME" = "UNIX" ]
```

So it reads:

```
if test $OS_NAME = "UNIX"
```
- 3** Save the file and exit the editor.
- 4** Test the script by executing it. Figure 7-16 shows sample output of the program.

A screenshot of a terminal window titled "Terminal". The window has a menu bar with "File", "Edit", "Settings", and "Help". The terminal content shows a user prompt "[jean@eli jean]\$./os_choice", followed by the script's output: "What is your favorite operating system? UNIX" and "You will like Linux.". The prompt returns to "[jean@eli jean]\$".

```
Terminal
File Edit Settings Help
[jean@eli jean]$ ./os_choice
What is your favorite operating system? UNIX
You will like Linux.
[jean@eli jean]$
```

Figure 7-16: Output of updated `os_choice` script

In the next exercise, you modify a while loop so it uses the test command.

To modify a while loop to use the test command:

- 1 Recall the script named `colors`, which you wrote in Chapter 6. It repeatedly asks the user to guess its favorite color, until the user guesses the color red.

```
echo -n "Try to guess my favorite color "
read guess
while [ "$guess" != "red" ]; do
    echo "No, not that one. Try again. "; read guess
done
```

- 2 Load the file into `vi` or `Emacs`.

- 3 Change the line that reads:

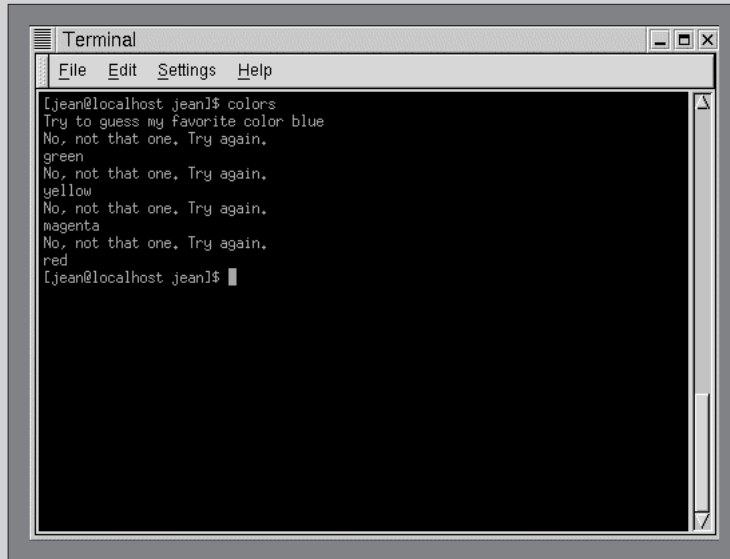
```
while [ "$guess" != "red" ]; do
```

So it reads:

```
while test $guess != "red" ; do
```

- 4 Save the file and exit the editor.

- 5 Test the script. Figure 7-17 shows sample output of the program.



```
Terminal
File Edit Settings Help
[jean@localhost jean]$ colors
Try to guess my favorite color blue
No, not that one. Try again.
green
No, not that one. Try again.
yellow
No, not that one. Try again.
magenta
No, not that one. Try again.
red
[jean@localhost jean]$
```

Figure 7-17: Output of updated colors script

So far, you used the test command from both the command line and from within script files. You also configured your home directory structure and your PATH variable so you can run your script files directly from the source directory. Your next step is to begin enhancing the corp_phones program by formatting record output.

Formatting Record Output

To format record output, use the translate utility. The **translate utility (tr)**, as you will recall from Chapter 5, changes the standard input (characters you type at the keyboard) character by character. The standard input can also be redirected with the < operator to come from a file rather than the keyboard. For example, the following command sends the contents of the counters file as input to the tr command. It converts lowercase characters to uppercase.

```
tr [a-z] [A-Z] < counters
```

By using the `|` operator, this command also works as a filter in cases where the input comes from the output of another UNIX command. For example, the following command sends the output of the `cat` command to `tr`:

```
cat names | tr -s ":" " "
```

This sample command sends the contents of the `names` file to `tr`. The `tr` utility replaces each occurrence of the `:` character with a space. The `tr` utility works like the `sed` command, except that `sed` changes the standard input string by string, not character by character.

You now use the `tr` utility to change lowercase characters to uppercase, as well as replace colon characters with spaces.

To format using the `grep` and `tr` commands:

- 1 Use the `grep` command to retrieve a record from the `corp_phones` file that matches the phone number 219-555-4501, and then pipe the output to `tr` to replace the colon characters in the record with space characters. Type the following command and then press **Enter**:

```
$ grep 219-555-4501 corp_phones | tr ':' ' '
```

Your screen looks similar to Figure 7-18.

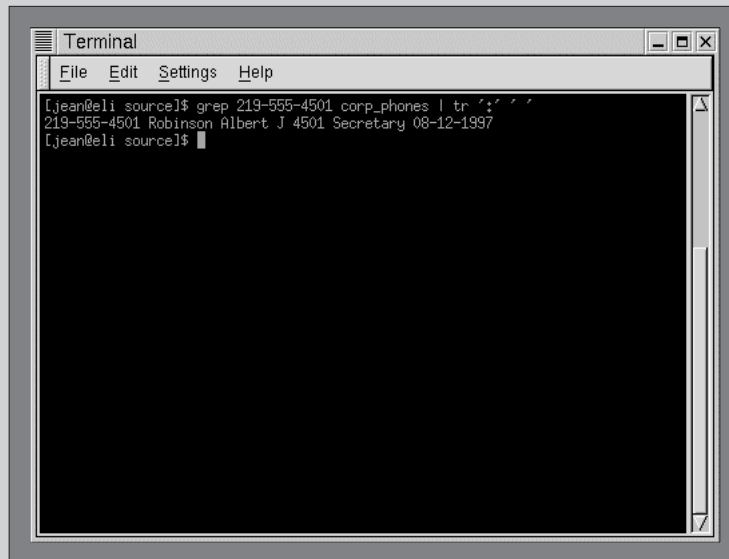


Figure 7-18: Output of `grep` and `tr` commands

- 2 Change lowercase characters to uppercase in the `corp_phones` file by entering the following command and then pressing **Enter**:

```
$ cat corp_phones | tr 'a-z' '[A-Z]'
```

Your screen looks similar to Figure 7-19.

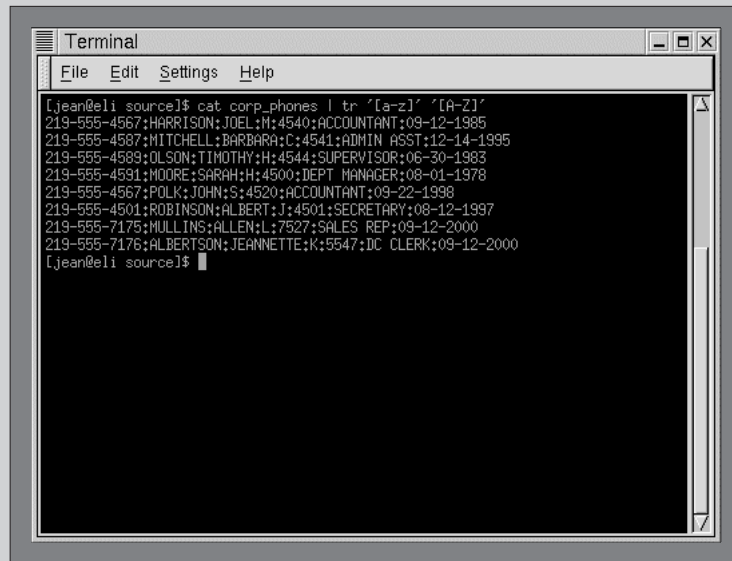


Figure 7-19: Output of `cat` and `tr` commands

To search for phone numbers in the `corp_phones` file, your program can use techniques similar to those you executed in the steps above.

To add record-searching capability to your program:

- 1 The `phmenu` program is already equipped to call the script `phonefind` when the user selects `S` from the menu. This command instructs the program to search for a phone number. Use the `vi` or `Emacs` editor to create the `phonefind` program. Type the code shown:

```
#!/bin/bash
#=====
# Script Name: phonefind
# By:          TEG
# Date         November 2000
# Purpose:     Searches for a specified record in the
#              corp_phones file
#=====
phonefile=~/.corp_phones
clear
cursor 5 1
echo "Enter phone number to search for: "
cursor 5 35
read number
echo
grep $number corp_phones | tr ':' ' '
echo
echo "Press ENTER to continue..."
read continue
```

- 2 Save the file and exit the editor.
- 3 Use the **chmod** command to make the file executable, and then test the script by searching for the number 219-555-7175. Your screen should look similar to Figure 7-20.

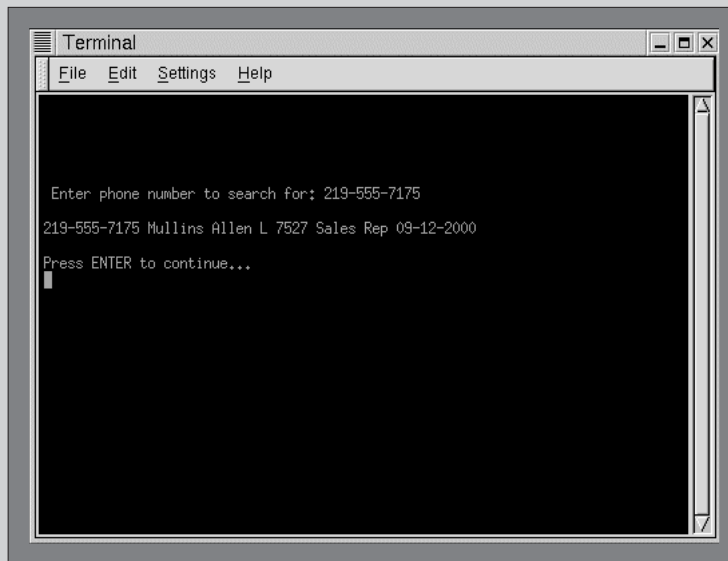


Figure 7-20: Output of phonefind script

You formatted the record output of the program; your next task is to delete phone records.

Deleting Phone Records

In this section you review the `sed` command. Recall from Chapter 5 that `sed` takes the contents of an input file and applies actions, provided as options and arguments, to the file's contents. The results are sent to the standard output device. A simple way to delete a phone record using `sed` is with the `d` (delete) option. Here is a pseudocode representation of the necessary steps:

Enter phone number

Use `sed -d` to delete the matching phone number and output to a temporary file, `f`

Confirm acceptance

If the output is accepted, copy the temporary file `f` back to `corp_phones` (overlying it)

Now you revise the `phmenu` script to include the delete option.

To delete phone records by editing the `phmenu` program:

- 1 Make sure you are in the source directory. Using the `vi` or Emacs editor, retrieve the revised `phmenu` program and add the code shown in boldface.

```
#!/bin/bash
# =====
# Script Name: phmenu
# By: JQD
# Date: November 1999
# Purpose: A menu for the Corporate Phone List
# Command Line: phmenu
# =====
phonefile=~/.source/corp_phones
loop="y"
while test $loop = "y"
do
clear
cursor 3 12; echo "Corporate Phone Reporting Menu"
cursor 4 12; echo "=====
cursor 6 9; echo "V - View Phone List"
cursor 7 9; echo "P - Print Phone List"
cursor 8 9; echo "A - Add Phone to List"
cursor 9 9; echo "S - Search for Phone"
cursor 10 9; echo "D - Delete Phone"
cursor 12 9; echo "Q - Quit"
```

```

cursor 12 32;
read choice || continue
case $choice in
    [Aa])  phoneadd ;;
    [Pp])  phlist1 ;;
    [Ss])  phonefind ;;
    [Vv])  clear ; less $phonefile ;;
    [Dd])  cursor 16 4; echo "Delete Phone Record"
           cursor 17 4; echo "Phone:"
           cursor 17 7; read number
           cursor 18 4; echo "Accept? (y)es or (n)o"
           cursor 18 26; read Accept
           if test $Accept = "y"
           then
               sed /$number/d$phonefile > f
               cp f $phonefile
           fi
           ;;
    [Qq])  clear; exit ;;
    *)     cursor 14 4; echo "Invalid Code"; read prompt ;;
esac
done

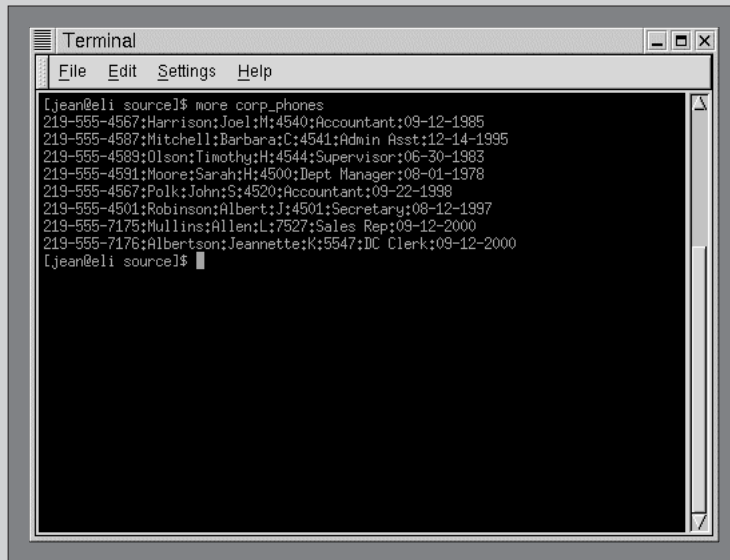
```

- 2** Save the file and exit the editor. Later, when you test the program, the menu will appear similar to Figure 7-21.



Figure 7-21: Output of phmenu script

- 3 Use the more command to display the contents of the file before you delete a record. Your screen should be similar to Figure 7-22.



```
Terminal
File Edit Settings Help

[jean@eli source]$ more corp_phones
219-555-4567:Harrison:Joel:M:4540:Accountant:09-12-1985
219-555-4587:Mitchell:Barbara:C:4541:Admin Asst:12-14-1995
219-555-4589:Olson:Timothy:H:4544:Supervisor:06-30-1983
219-555-4591:Moore:Sarah:H:4500:Dept Manager:08-01-1978
219-555-4567:Polk:John:S:4520:Accountant:09-22-1998
219-555-4501:Robinson:Albert:J:4501:Secretary:08-12-1997
219-555-7175:Mullins:Allen:L:7527:Sales Rep:09-12-2000
219-555-7176:Albertson:Jeannette:K:5547:DC Clerk:09-12-2000
[jean@eli source]$
```

Figure 7-22: Contents of corp_phones file

- 4 Run the phmenu program and test the delete option by removing phone number 219-555-4567. Figure 7-23 shows the Delete Phone Record screen.



```
Terminal
File Edit Settings Help

Corporate Phone Reporting Menu
=====

V - View Phone List
P - Print Phone List
A - Add Phone to List
S - Search for Phone
D - Delete Phone

Q - Quit          d

Delete Phone Record
Phone:219-555-5671
Accept? (y)es or (n)o
```

Figure 7-23: Delete Phone Record screen

- 5 Type **y** to confirm the deletion.
- 6 On the Main menu, select option **V** to view the phone file. Your screen looks similar to Figure 7-24.

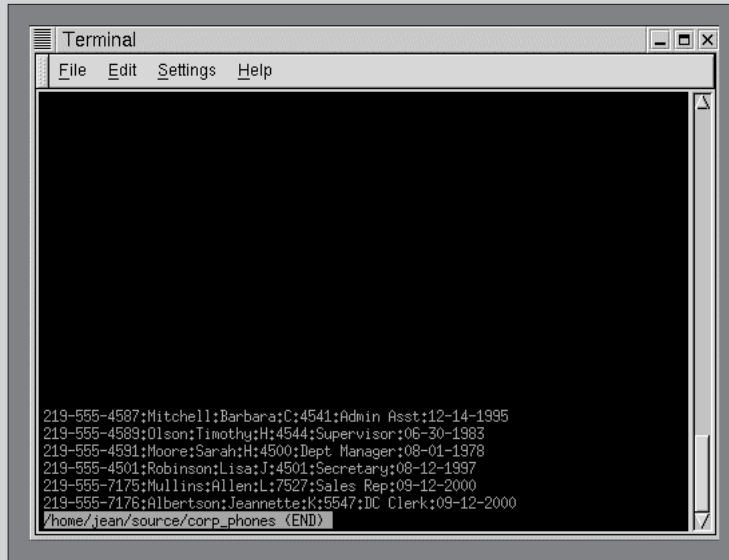


Figure 7-24: View Phone List option

Notice the record for phone number 219-555-4567 is no longer in the file.

In this section you began to revise the shell phone program to meet Dominion Consulting's needs. You added code for the delete option to the phmenu script. In the next section you add code that prevents duplicate phone numbers from being entered in the file.

S U M M A R Y

- The two most popular and proven analysis tools are the program flowchart and pseudocode. The flowchart is a logic diagram drawn using a set of standard symbols that explain the flow and the action to be taken by the program.
- Pseudocode is a model of a program. It is written in statements similar to actual code.
- You can use the first line in a script file to tell the operating system which shell to use when interpreting the script.

- You can use the test command to validate the existence of directories and files as well as compare numeric and string values. You can place it inside your shell script or execute it directly from the command line.
- The translate utility (tr) changes the characters typed at the keyboard, character by character, and also works as a filter when the input comes from the output of another UNIX command. Standard input can also be redirected to come from a file rather than the keyboard.
- The sed command reads a file as its input and outputs the file's modified contents. You specify options and pass arguments to sed to control how the file's contents are modified.



REVIEW QUESTIONS

1. Pseudocode is written to _____.
 - a. guide the programmer in writing the program
 - b. help eliminate the chance of omitting necessary instructions in the program
 - c. match, verbally, the symbolic representation in the flowchart
 - d. All of the above
2. To tell the system to use the Bash shell to interpret a script, use this code as the first line of the script file _____.
 - a. `!/bin/bash`
 - b. `#/bin/bash`
 - c. `#!/bin/bash`
 - d. `#$/bin/bash`
3. The test command compares two numeric values using the _____ option operator.
 - a. `=`
 - b. `string`
 - c. `==`
 - d. `-eq`
4. The test command compares two string values using the _____ option operator.
 - a. `=`
 - b. `string`
 - c. `==`
 - d. `-eq`
5. The test command's `-a` option is the _____ operator.
 - a. logical AND
 - b. absolute value
 - c. test for hidden files
 - d. logical negation

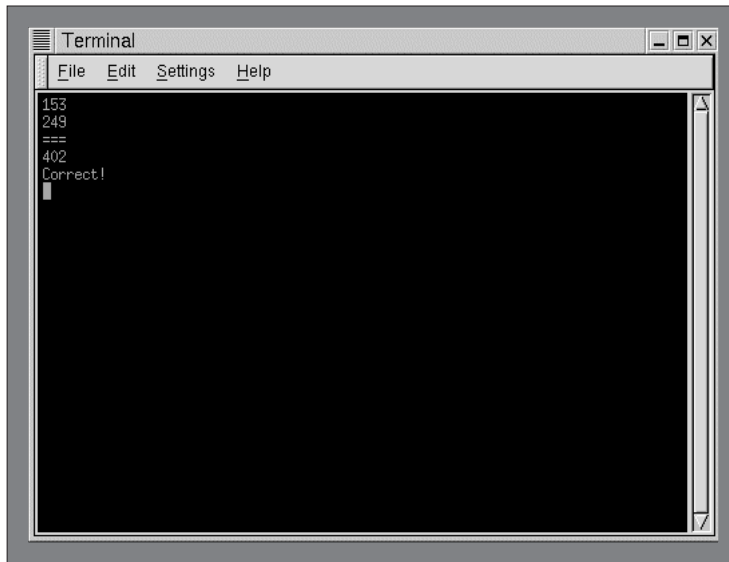
6. The test command's `-o` option is the _____ operator.
 - a. logical AND
 - b. logical OR
 - c. overwrite
 - d. logical negation
7. The `tr` command _____.
 - a. translates only what is typed at the keyboard
 - b. translates on a line basis like `sed`
 - c. translates on a character-by-character basis
 - d. translates characters originating from within files only
8. The `tr` command to change phone records to contain all lowercase characters is _____.
 - a. `tr upper lower corp_phones`
 - b. `tr [a-z] [A-Z] corp_phones`
 - c. `cat corp_phones | tr [a-z] [A-Z]`
 - d. `tr [A-Z] [a-z] < corp_phones`
9. The test command returns a value of _____ to indicate TRUE and _____ to indicate FALSE.
 - a. 0, 1
 - b. 1, 0
 - c. blank err
 - d. OK, !OK
10. The test command's return value appears when you type _____ at the command line.
 - a. `echo $$`
 - b. `echo $?`
 - c. `echo $-`
 - d. `echo $.`



E X E R C I S E S

1. Use the `tr` command to translate the phone records file to contain a hyphen as the field separator instead of a colon (:). (Be sure to save the translated records to a different file.)
2. Set an environment variable, `RHL`, to the value "Red Hat Linux," and use the test and `echo $?` commands to determine if it matches "Red Hat Linux" and "RED HAT LINUX." How can you tell which one matches?
3. Set an environment variable called `NUMVAL` to 2000, and use the test and `echo $?` commands to determine if it matches the value 2000. What about 2020? How can you tell which one matches?

4. Write a flowchart for a program that tests your ability to perform simple arithmetic. The program should display two numbers that are to be added and ask the user to enter the sum. If the correct answer is given, the program should congratulate the user. If an incorrect answer is given, the program should show the user the correct answer.
5. Write pseudocode for the flowchart program you created in Exercise 4.
6. Write the actual code for the math-testing program you flowcharted in Exercise 4 and developed pseudocode for in Exercise 5. Figure 7-25 shows an example of the program's screen output.



```
Terminal
File Edit Settings Help
153
249
===
402
Correct!
█
```

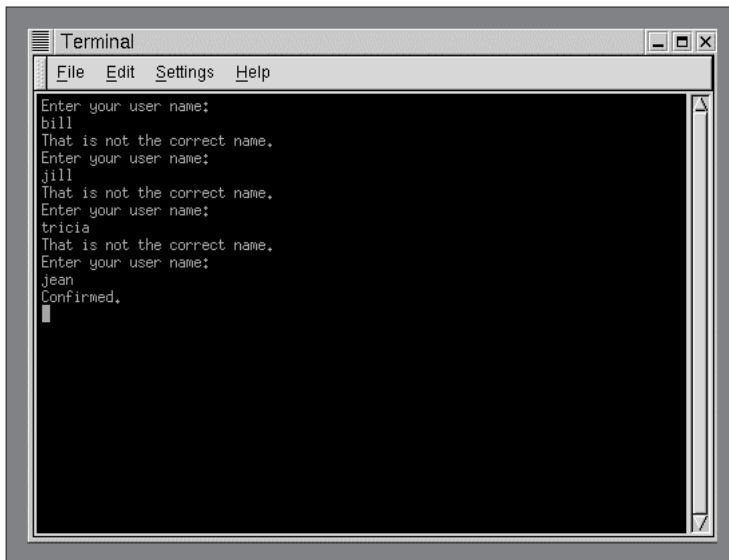
Figure 7-25: Math-testing program

7. Use vi or Emacs to create the file `My_old_cars`. The file should contain these records:
1948:Ford:sedan
1952:Chevrolet:coupe
1960:Ford:Mustang
1972:Chevrolet:Corvette
1977:Plymouth:Roadrunner

Next, write a shell script that displays a data entry screen. The script should allow additional records to be entered into the file.
8. Write a shell script that displays all records in the `My_old_cars` file. The colons should be converted to spaces before the output appears.

 **DISCOVERY EXERCISES**

1. Sort the `corp_phones` file by last name, and pipe the output to a `tr` command to convert all characters to uppercase and store output in a new file, `Phoneupper`.
2. Create a directory called `Scripts` in your source directory, and use the `test` command to validate that the directory exists.
3. Use your choice of the `sed` or `tr` command to convert all occurrences of the word “Accountant” in the `corp_phones` file to the word “Bookkeeper.”
4. User Authentication Program: UNIX keeps users’ login name stored in an environment variable named `USER`. Design a flowchart for a program that asks users who they are. Their response should be compared against the contents of the `USER` variable. If the two strings do not match, users should be asked again. This process should repeat until the correct user name is entered.
5. User Authentication Program: Write the pseudocode for the flowchart you designed in Discovery Exercise 4.
6. User Authentication Program: Write the actual code for the program you flowcharted in Discovery Exercise 4 and wrote pseudocode for in Discovery Exercise 5. Figure 7-26 shows a sample of the program’s output.



```
Terminal
File Edit Settings Help
Enter your user name:
bill
That is not the correct name.
Enter your user name:
jill
That is not the correct name.
Enter your user name:
tricia
That is not the correct name.
Enter your user name:
jean
Confirmed.
█
```

Figure 7-26: Output of User Authentication Program

LESSON B

objectives

In this lesson you will:

- Create a program algorithm to solve a problem
- Learn to create a quick screen-clearing technique
- Develop and test a program to re-enter fields during data entry
- Develop and test a program to eliminate duplicate records
- Develop and test a program to delete records from the file
- Create shell functions
- Load shell functions automatically from the command line when you log on

Completing the Case Project

Clearing the Screen

Before you begin to complete the programming tasks, you decide you want to clear screens more quickly. The **clear** command is a useful housekeeping utility for clearing the screen before new screens appear (which happens frequently in shell scripts), but you can use a faster method. You can store the output of the clear command in a shell variable. Recall from Chapter 6 that you can store the output of a command in a variable by enclosing the command in single back quotes. For example, this command stores the output of the date command in the variable TODAY:

```
TODAY='date'
```

The output of the clear command is a sequence of values that erases the contents of the screen. Storing these values in a variable and then echoing the contents of the variable on the screen accomplishes the same thing, but about ten times faster. This is because the system does not have to locate and execute the clear command, as it does when executing the clear command.

To clear screens by setting a shell variable:

- 1 Set a shell variable, CLEAR, to the output of the clear command, by typing:

```
CLEAR='clear'  
export CLEAR
```

- 2 Use your new variable in your shell programs for a fast clear operation:

```
echo "$CLEAR"
```

- 3 To make this fast clear always available, place these two lines of code at the end of your `.bashrc` file (or the equivalent login initialization file for your shell):

```
CLEAR='clear'
export CLEAR
```

- 4 Save the file, exit, and then log on again to activate the login script.

You are now ready to complete your first task: correcting entries. To do this, you need to manipulate the cursor.

Moving the Cursor

The first task is to let the user return to a previously entered field and to correct data in that field before continuing. Recall that in Chapter 6 you created a script named `cursor`. The script accepts two arguments: the row and column number of a screen position. The `cursor` script then positions the cursor at the specified position using the `tput` command.

You want to allow users to return the cursor to a previous field on the screen when adding records to the `corp_phones` file. You decide that the minus character (-) will signal this. If the user enters a minus and presses the Enter key, the cursor is repositioned at the start of the previous field as shown on the screen. You will make this change by editing the `phoneadd` program (which provides the user with data-entry screens) that you already created. Your first step in editing the program is to create a program algorithm.

Creating Program Algorithms

An **algorithm** is a sequence of commands or instructions that produces a desired result. The algorithm to solve a specific problem is frequently developed by following the logic shown in a flowchart and the expressed conditions necessary to carry out the logic described in the pseudocode.

Here is the pseudocode for repositioning the cursor at the previous field when the user enters the minus sign (-):

```
Read information into field2.
While field2 equals "-"
    Move cursor to position of previous field, field1.
    Clear current information displayed in field1.
    Read new information into field1.
    If field1 = "q"
    then
        Exit program.
```

```

End if.
Move cursor to position of field2.
Read information into field2.
End While.

```

One code addition to the phoneadd program uses the algorithm for re-entering fields:

```

cursor 5 18; read lname
while test "$lname" = "-"
do      cursor 4 18; echo "
        cursor 4 18; read phonenum
        cursor 5 18; read lname
done

```

This code reads the last name into the variable lname. If lname contains a minus sign (-), the cursor moves to the previous field, which contains the phone number. The value displayed for the phone number is cleared from the screen, and a new value is entered into phonenum. The cursor is then moved back to the last name field, and the last name is entered. The while statement repeats this process as long as the user types a minus sign for the last name.

Note: Using the if statement instead of the while statement allows only one return to the prior field. Instead, you need a loop so the process repeats as long as the user enters a minus sign for the field.

Look at the while statement shown above:

```
while test "$lname" = "-"
```

The argument “\$lname” is enclosed in quotation marks to prevent the command from producing an error, in the event the user types more than one word for the last name. For example, if the user enters Smith Williams for the last name, the statement above will be interpreted as:

```
while test "Smith Williams" = "-"
```

However, if the statement were written without the quotation marks around \$lname, the statement would be interpreted as:

```
while test Smith Williams = "-"
```

This statement causes an error message, because it passes too many arguments to the test command.

You are now ready to edit the phoneadd program. You add the field re-entering algorithm to each part of the program that reads a value into a field.

To allow re-entry of data:

- 1 Make sure you are in the source directory. Load the Phoneadd program into the vi or Emacs editor.

- 2** Add the boldfaced code shown below to the program. Notice that the revised code also includes your new, faster, screen clear feature. It also changes the existing if statements, so they use the test command.

```
#!/bin/bash
# =====
# Script Name: phoneadd
# By:          JQD
# Date:        March 1999
# Purpose:     A shell script that sets up a loop to add
#              new employees to the phonelist file.
#              The code also prevents duplicate phone
#              numbers
#              from being assigned.
# Command Line:  phoneadd
# =====
trap "rm ~/tmp/* 2> /dev/null; exit" 0 1 2 3
phonefile=~/source corp_phones
looptest="y"
while test "$looptest" = "y"
do
    clear
    cursor 1 4 ; echo "Corporate Phone List Additions"
    cursor 2 4 ; echo "===== "
    cursor 4 4 ; echo "Phone Number: "
    cursor 5 4 ; echo "Last Name   : "
    cursor 6 4 ; echo "First Name  : "
    cursor 7 4 ; echo "Middle Init : "
    cursor 8 4 ; echo "Dept #      : "
    cursor 9 4 ; echo "Job Title   : "
    cursor 10 4; echo "Date Hired  : "
    cursor 12 4; echo "Add another? (y)es or (q)uit "
    cursor 4 18; read phonenum
    if test $phonenum = "q"
    then
        clear ; exit
    fi
    cursor 5 18 ; read lname
    while test "$lname" = "-"
    do
        cursor 4 18 ; echo " "
        cursor 4 18 ; read phonenumber
        if test "$phonenum" = "q"
        then
            clear ; exit
        fi
        cursor 5 18 ; read lname
    done
done
```

```

cursor 6 18 ; read fname
while test "$fname" = "-"
do
    cursor 5 18 ; echo "
        cursor 5 18 ; read lname
        if test "$lname" = "q"
        then
            clear ; exit
        fi
    cursor 6 18 ; read fname
done
cursor 7 18 ; read midinit
while test "$midinit" = "-"
do
    cursor 6 18 ; echo "
        cursor 6 18 ; read fname
        if test "$fname" = "q"
        then
            clear ; exit
        fi
    cursor 7 18 ; read midinit
done
cursor 8 18 ; read deptno
while test "$deptno" = "-"
do
    cursor 7 18 ; echo "
        cursor 7 18 ; read midinit
        if test "$midinit" = "q"
        then
            clear ; exit
        fi
    cursor 8 18 ; read deptno
done
cursor 9 18 ; read jobtitle
while test "$jobtitle" = "-"
do
    cursor 8 18 ; echo "
        cursor 8 18 ; read deptno
        if test "$deptno" = "q"
        then
            clear ; exit
        fi
    cursor 9 18 ; read jobtitle
done
cursor 10 18 ; read datehired
while test "$datehired" = "-"

```

```

do
    cursor 9 18 ; echo "
    cursor 9 18 ; read jobtitle
    if test "$jobtitle" = "q"
    then
        clear ; exit
    fi
    cursor 10 18 ; read datehired
done
#Check to see if last name is not blank before you
#write to disk
if test "$lname" != ""
then
    echo "$phonenumber:$lname:$fname:$midinit:$deptno:
$jobtitle:$datehired" >> $phonefile
fi
cursor 12 33 ; read looptest
if test "$looptest" = "q"
then
    clear ; exit
fi
done

```

- 3** Save the file and exit the editor.
- 4** Execute the phoneadd script. For the phone number, enter **219-555-4523** and press **Enter**. Your screen appears similar to Figure 7-27.

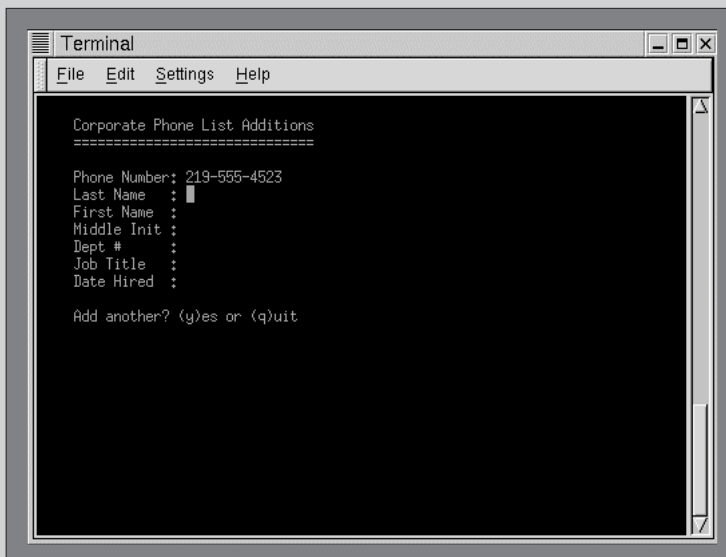


Figure 7-27: phoneadd screen

- 5** In the Last Name field, type the **minus sign character (-)** and press **Enter**. Your cursor moves back the Phone Number field. Your screen looks like Figure 7-28.

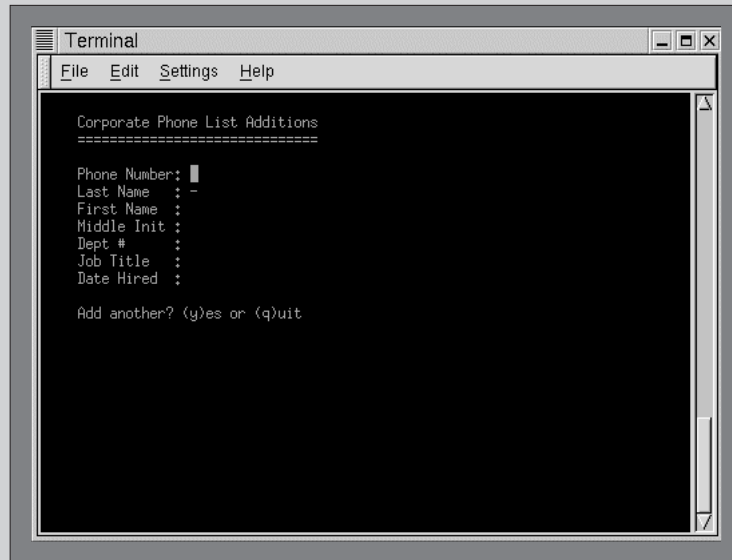


Figure 7-28: Phoneadd screen ready for data re-entry

- 6** Re-enter the phone number as **219-555-4511**, and press **Enter**.
- 7** Complete the remaining fields with the information shown below. As the cursor moves to each field, test the program by typing the **minus sign** and pressing **Enter**. The cursor should move to the previous field each time.
- Last name: Brooks**
First name: Sally
Middle initial: H
Department Number: 4540
Job Title: Programmer
Date Hired: 11-23-1999
- 8** After you enter all the information, quit the program. Use the `cat` command to display the contents of the `corp_phones` file. The new record should appear.

You completed the first task. You are now ready to work on the second task: protecting against duplicate phone numbers.

Protecting Against Entering Duplicate Phone Numbers

Because users do not always enter valid data, a program should always check its input to ensure the user has entered acceptable information. This is known as **input validation**. Your next task is to create an input validation algorithm that prevents the user from adding a phone number that has already been assigned. The pseudocode and flowchart to accomplish this are shown in Figure 7-29.

Pseudocode:

```
If phone number is already on file
    then
        display message on the screen "This number has already been assigned to:"
        display the person's record who has the duplicate number.
        Clear the screen and prepare for another entry
    End if
```

Flowchart:

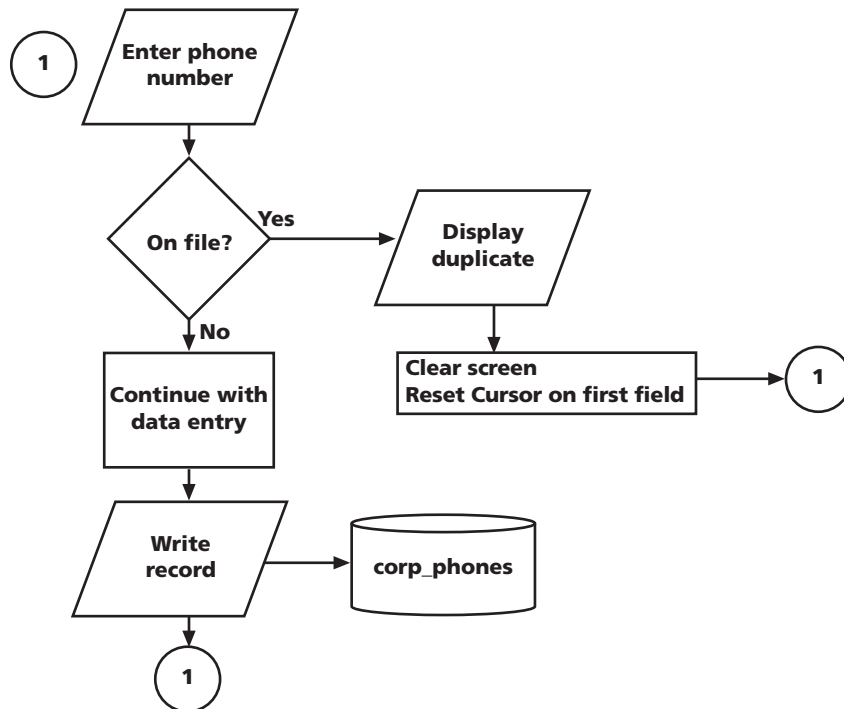


Figure 7-29: Revision #2 to phoneadd program script

In the following steps, you will add input validation code to the phoneadd script.

To prevent phone number duplications:

- 1** Make sure you are in the source directory. Load the phoneadd script into vi or Emacs.
- 2** Add the boldfaced section of code to complete the revised script.

```
#!/bin/bash
#=====
# Script Name: phoneadd
# By:          JQD
# Date:        March 1999
# Purpose:     A shell script that sets up a loop to add
#              new employees to the phonelist file.
#              The code also prevents duplicate phone
#              numbers
#              from being assigned.
# Command Line:  phoneadd
# =====
trap "rm ~/tmp/* 2> /dev/null; exit" 0 1 2 3
phonefile=~/.corp_phones
looptest=y
while test "$looptest" = "y"
do
    clear
    cursor 1 4 ; echo "Corporate Phone List Additions"
    cursor 2 4 ; echo "=====
    cursor 4 4 ; echo "Phone Number: "
    cursor 5 4 ; echo "Last Name   : "
    cursor 6 4 ; echo "First Name  : "
    cursor 7 4 ; echo "Middle Init : "
    cursor 8 4 ; echo "Dept #      : "
    cursor 9 4 ; echo "Job Title   : "
    cursor 10 4; echo "Date Hired  : "
    cursor 12 4; echo "Add another? (y)es or (q)uit "
    cursor 4 18; read phonenum
    if test $phonenum = "q"
    then
        clear ; exit
    fi
```

```

# Check to see if the phone number already exists
while grep "$phonenum" $phonefile > ~/tmp/temp
do
cursor 19 1 ; echo "This number has already been assigned to:"
    cursor 20 1 ; tr ':' ' ' < ~/tmp/temp
    cursor 21 1 ; echo "Press ENTER to continue... "
    read prompt
    cursor 4 18 ; echo "
    cursor 4 18 ; read phonenum
done
cursor 5 18 ; read lname
... The remainder of the program is unchanged

```

- 3** Save the file and exit the editor.
- 4** If you have not already created a tmp directory under your home directory, do so now.
- 5** Run the program. Test it by entering a phone number that already exists in the file, such as **219-555-4587**. Your screen should look like Figure 7-30.

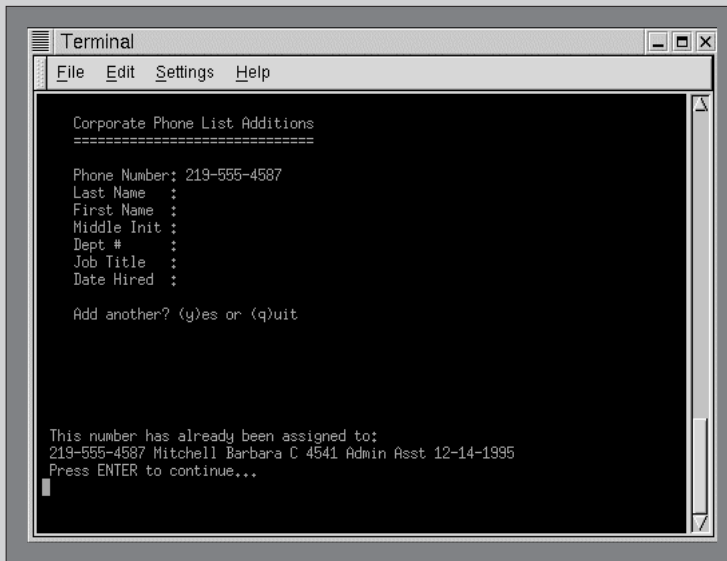


Figure 7-30: Revised phoneadd program

- 6** Complete the data entry screen by entering a valid phone number.

Looking at the `phoneadd` program, you realize that it contains code that you may want to reuse for other programs. To do this, you use shell functions.

Using Shell Functions

A **shell function** is a group of commands that is stored in memory and assigned a name. Shell scripts can use the function name to execute the commands. You can use shell functions to isolate reusable code sections, so that you do not have to duplicate algorithms throughout your program. For example, the `phlist1` script can use several functions to sort the phone list in a variety of ways.

A function name differs from a variable name, because a function name is followed by a set of parentheses, and the commands that comprise the function are enclosed in curly braces. For example, look at the code for a function:

```
datenow()
{
  date
}
```

The name of the function is “`datenow`.” It has only one command inside its braces: the `date` command. When the `datenow` function is executed, it calls the `date` command.

Functions are usually stored in script files and loaded into memory when you log on. However, you can also enter them at the command line.

To declare the simple `datenow` function:

- 1** At the command line, type **`datenow()`** and press **Enter**. Notice the prompt changes to the `>` symbol. This indicates the shell is waiting for you to type more information to complete the command you started.
- 2** At the `>` prompt, type **`{`** and press **Enter**.
- 3** At the `>` prompt, type **`date`** and press **Enter**.
- 4** At the `>` prompt type **`}`** and press **Enter**. The normal prompt returns.
- 5** You created the `datenow` function and stored it in the shell’s memory. Invoke it by typing **`datenow`** and pressing **Enter**. Your screen looks similar to Figure 7-31.

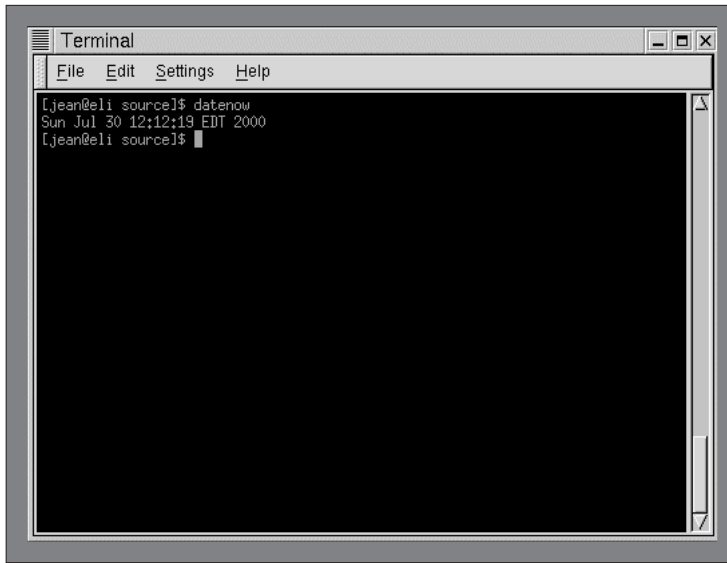


Figure 7-31: Output of datenow function

Arguments are passed to functions in the same manner as any other shell procedure. The function accesses the arguments using the positional variables `$1 . . . $9`. Simply type the arguments following the command name, placing a space between each argument. Now you redefine the `datenow` function so it accepts an argument.

To redefine the `datenow` function to accept an argument:

- 1** At the command line, type **`datenow()`** and press **Enter**. The commands you are about to type replace those currently stored in the `datenow` function.
- 2** At the `>` prompt, type **`{`** and press **Enter**.
- 3** At the `>` prompt, type **`echo "$1"`** and press **Enter**. When the function runs, this command displays the information passed to the function in the first argument.
- 4** At the `>` prompt, type **`date`** and press **Enter**.
- 5** At the `>` prompt type **`}`** and press **Enter**. The normal prompt returns.
- 6** Test the function by typing **`datenow "Today is"`** and pressing **Enter**. Your screen looks similar to Figure 7-32.

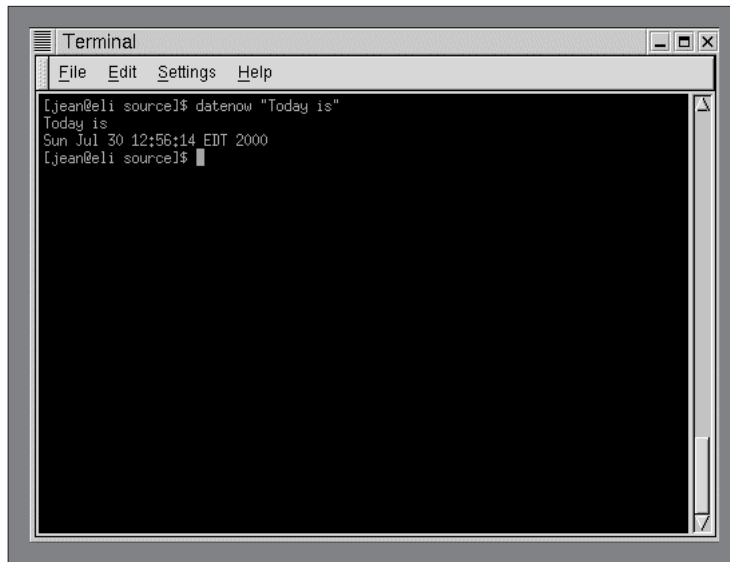


Figure 7-32: Results of revised datenow function

The exercises you just completed demonstrate how functions work in general. Typing functions at the command line is hardly productive, however, because they must be re-entered each time you log on. You will now learn how to use shell functions to reuse code and to make functions available each time you log in.

Reusing Code

To improve your programming productivity, you should learn to reuse your code. This means that the functions and programs you develop should be shared with other programs and functions as much as possible, thereby helping to prevent duplications, save time, and reduce errors. A good illustration of this potential use of reusable functions can be demonstrated in the current phoneadd program. You can create several different sort functions and store them in memory. The phlist1 script can then call these functions to display the list of phone numbers, sorted in a variety of ways.

You can place multiple functions inside a shell script such as `.myfuncs` and execute them from your `.bash_profile` login script or simply run `.myfuncs` from the command line. This loads all your functions into memory just as you load environment variables.

To place several functions inside a shell script:

- 1 Use the vi or Emacs editor to create the `.myfuncs` file inside your source directory.
- 2 Enter the code for the following functions:

```
sort_name()  
{  
    sort +1 -t: corp_phones  
}  
sort_date()  
{  
    sort +6 -t: corp_phones  
}  
sort_dept()  
{  
    sort +4 -t: corp_phones  
}
```

- 3 Save the file and exit the editor.

Your next task is to load the `.myfuncs` file into memory so its functions may be executed. To do this, type a period (`.`) followed by a space, followed by the name of the file containing the functions.

To load the `.myfuncs` file:

- 1 At the command line, type `..myfuncs` and press **Enter**. Nothing appears, but the functions are loaded into memory.
- 2 Test some functions. Type `sort_name` and press **Enter**. You see the phone records sorted by individuals' names.
- 3 Type `sort_dept` and press **Enter**. You see the phone records sorted by department number.

You can load functions automatically by your `.bash_profile` or `.bashrc` files when you log on. This way, they are always available to any shell script that needs them.

To modify your `.bashrc` file to load the `.myfuncs` script:

- 1 Load your `.bashrc` file into the vi or Emacs editor.
- 2 At the end of the file, add the following command:

```
. ~/source/.myfuncs
```

- 3** Save the file and exit the editor.
- 4** Log out and log back on to load the functions.
- 5** Test the `sort_name`, `sort_dept`, and other functions.

Your last task is to display the phone listing in sorted order by employees' last names. You can do this by using your `sort_name` function, as stored in the `.myfuncs` file.

Sorting the Phone List

To sort the phone list, make a minor revision to `phlist1` to load the functions and then call `sort_name` to redirect the sorted output to a temporary file. The sorted temporary file serves as input to the Awk program that displays the records. The revised code will use the `CLEAR` variable also.

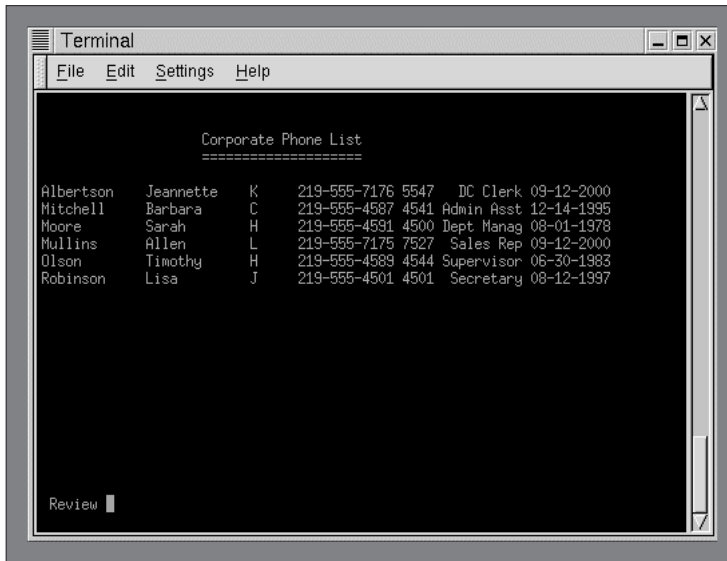
To sort the phone listing:

- 1** Make sure you are in the source directory. Use the `vi` or Emacs editor to add the code to the `phlist1` script. The additions and revisions are in boldface.

```
#!/bin/bash
#=====
# Script Name: phlist1
# By:      JQD
# Date:    March 1999
# Purpose: Use awk to format a colon-separated set of fields
#          in a flat file and display to the screen
# Command line:    bash phlist <enter>
# =====
echo "$CLEAR"
cursor 2 20; echo "Corporate Phone List"
cursor 3 20; echo "=====
cursor 5 0;
sort_name > sorted_phones
awk -F: ' { printf "%-12s %-12s %s\t%s%s%10.10s  %s\n",
                $2, $3, $4, $1, $5, $6, $7 } ' sorted_phones

cursor 23 1; echo "Review"
cursor 23 8; read prompt
```

- 2** This code assumes you have modified your `.bashrc` file so it loads the functions in `.myfuncs` when you log on. Save the file and exit the editor.
- 3** Test the file by typing **phlist1** and pressing **Enter**. Your screen looks similar to Figure 7-33. Press **Enter** when you finish observing the screen.



```

Terminal
File Edit Settings Help

Corporate Phone List
=====
Albertson   Jeannette  K    219-555-7176 5547  IC Clerk 09-12-2000
Mitchell    Barbara    C    219-555-4587 4541  Admin Asst 12-14-1995
Moore       Sarah      H    219-555-4591 4500  Dept Manag 08-01-1978
Mullins     Allen      L    219-555-7175 7527  Sales Rep 09-12-2000
Olson       Timothy    H    219-555-4589 4544  Supervisor 06-30-1983
Robinson    Lisa       J    219-555-4501 4501  Secretary 08-12-1997

Review

```

Figure 7-33: Output of revised phlist1 script

Adding code to call the other sort functions will be an exercise at the end of this chapter.

In this chapter you have learned to plan algorithms and programs using flowcharts and pseudocode. You have also learned to create complex decision expressions with the test command. You have furthered your use of the grep, tr, and sed commands to format output. In addition, you have learned advanced programming techniques, such as repositioning the cursor at a previous field in a data entry screen, and creating shell functions.

S U M M A R Y

- To speed clearing the screen, assign the clear command sequence to the shell variable CLEAR that can be set inside your login script, .bashrc. This clears your screen faster because it does not require a look-up sequence in a file every time it executes.
- An algorithm is a sequence of instructions or commands that produce a desired result. Following the logic flow expressed in flowcharts and pseudocode develops algorithms.
- Shell functions can simplify the program code by isolating code that can be reused throughout that program as well as others.

COMMAND SUMMARY

Options and arguments of test command

Option	Meaning	Example Command
-eq	equal to	test a -eq b
-gt	greater than	test a -gt b
-lt	less than	test a -lt b
-ge	greater than or equal to	test a -ge b
-le	less than or equal to	test a -le b
-ne	not equal to	test a -ne b
-z	Tests for a zero-length string	test -z string
-n	Tests for a non-zero string length	test -n string
string1 = string2	Tests two strings for equality	test string1 = string2
string1 != string2	Tests two strings for inequality	test string1 != string2
string	Tests for a non-zero string length	test string
-e	True if a file exists	test -e file
-r	True if a file exists and is readable	test -r file
-w	True if a file exists and is writeable	test -w file
-x	True if a file exists and is executable	test -x file
-d	True if a file exists and is a directory	test -d file
-f	Tests if a file exists and is a regular file	test -f file
-s	True if a file exists and has a size greater than zero	test -s file
-c	Tests if a file exists and is a character special file (which is a character-oriented device, such as a terminal or printer)	test -c file
-b	Tests if a file exists and is a block special file (which is a block-oriented device, such as a disk or tape drive)	test -b file
-a	Logical AND	test expression1 -a expression2
-o	Logical OR	test expression1 -o expression2
!	Logical negation	test !expression

REVIEW QUESTIONS

1. An algorithm is _____.
 - a. a name that replaces the standard UNIX command name
 - b. an alternate name for a UNIX command
 - c. always entered into the system-wide initialization file, /etc/profile
 - d. a derived formula made up of a sequence of commands that produce a desired result
2. A shell function can be recognized because it has _____ after the function name.
 - a. curly braces
 - b. parentheses
 - c. brackets
 - d. quotation marks
3. A shell function's commands are enclosed inside _____.
 - a. curly braces
 - b. parentheses
 - c. brackets
 - d. quotation marks
4. Shell functions are useful because _____.
 - a. you can reuse the code in other programs
 - b. it cuts down on re-entry of duplicate code inside shell programs
 - c. increases your productivity as a shell programmer
 - d. all of the above
5. What is the proper sequence for developing an algorithm?
 - a. flowchart—algorithm—pseudocode
 - b. flowchart—pseudocode—algorithm.
 - c. algorithm—flowchart—pseudocode
 - d. there is no one correct sequence
6. How do you pass arguments to a shell function?
 - a. Enclose them inside the parentheses that follow the function name.
 - b. Use the positional variables, \$1, \$2, \$3 .. \$9.
 - c. Use the \$? shell variable.
 - d. Use the \$# shell variable.
7. The output of a command can be stored in a shell variable by enclosing the command in _____.
 - a. parentheses
 - b. double quotation marks
 - c. single quotation marks
 - d. back quote marks

8. Writing code to test and determine if user input is correct is known as _____.
 - a. algorithm design
 - b. data validation
 - c. flowcharting
 - d. pseudocode design
9. If `.numeric_functions` is a script file of shell functions, the _____ command is used to load it into memory.
 - a. `numeric_functions`
 - b. `load numeric_functions`
 - c. `.numeric_functions`
 - d. `install numeric_functions`
10. True or false: shell functions may be executed from the command line.



E X E R C I S E S

1. From the command line create the function `dir` to execute this: `ls -lq | more`.
2. From the command line create the function `simple_date` to execute this: `date +%D`.
3. Create the executable script file `.mystuff` to contain the two functions created in Exercises 1 and 2, and to call each function from the command line.
4. In the Lesson A exercises, you created the file `my_old_cars`. Write a shell script that allows the user to enter a search string and displays all records in the file that contain a string matching the search string. For example, if the user enters “Ford,” the script displays all records that contain the word “Ford.”
5. Write a shell script that allows the user to enter a search string and deletes all records in the `my_old_cars` file that contain a string matching the search string. For example, if the user enters “1948,” the script deletes all records containing “1948.”
6. Create shell functions that sort the records in the file `my_old_cars`. Your functions should sort the records by year model (the first field), by make (the second field), and by car model (the third field). Write a menu program that calls each sort function.
7. Write a Main menu program for the `my_old_cars` file and all the script files you have created so far. Here is a list of the script files you should have:
 - Data entry screen
 - Script that displays all records in the file
 - Script that searches for and displays records containing a string
 - Script that deletes specified records from the file
 - Script that displays the records sorted in various ways



D I S C O V E R Y

1. Remove the colon characters from the `phoneadd` program and reposition the cursor to improve the appearance of the data entry screen.
2. Place the screen display from the `phoneadd` program in a function, called `phscreen`. Using the `vi` editor, type the function into `.mystuff` (from Exercise 3) and test the function by calling it from the command line.
3. Add menu items to `phmenu` that call the `sort_date` and `sort_dept` functions you created in the `.myfuncs` file. Test the program.
4. Create a shell function, `average`. Store it in a file, `avgfunc`. The function should accept three arguments. Assume the arguments are numbers. The function should calculate and output the average of the three numbers.
5. Create a script file, `minmax`. In the file, create a shell script, `min`. It should accept two arguments, assumed to be numbers. The function is to output the lesser of the two arguments. For example, if 5 and 10 are passed to the function, it should output 5.
6. Edit the `minmax` file you created in Discovery Exercise 6. Add a shell function named `max`. It should accept two arguments, assumed to be numbers. The function is to output the greater of the two arguments. For example, if 8 and 4 are passed to the function, it should output 8.